Project File System Design

Group 43

Name	Autograder Login	Email
Sarvagya Somvanshi	student176	sarvagya@berkeley.edu
Nathan Canera	student282	nathancanera@berkeley.edu
John Chang	student271	jchang04@berkeley.edu
Shengxiang Lin	student35	linshengxiang@berkeley.edu

Buffer Cache

Data Structures and Functions

```
struct buffer_cache {
  cached_block cached_blocks[64];
  int clock_hand;
  struct lock cache_lock; // only one thread can change the buffer cache's state
};


struct cached_block {
  block_sector_t sector;
  int use;
  int dirty;
  uint32_t* data;
  struct lock block_lock; // only one thread can read / write to block at a time
  struct list_elem elem;
};
```

```
struct inode {
  struct list_elem elem;
  block_sector_t sector;
  int open_cnt;
  bool removed;
  int deny_write_cnt;
};
```

Algorithms

`byte_to_sector`
Return `(inode->sector + 1) + pos / BLOCK_SECTOR_SIZE` (we no longer use the `inode_disk` struct when calculating sector).

`inode_create`
We modify this function so that there is no in-memory copy of the on-disk inode. In order to keep the metadata stored in `inode->data` in the starter code, we store the metadata in the `inode` itself, and we don't maintain a data attribute in our updated `inode` struct.

`inode_read_at`
This function iterates through each sector that contains the inode's data and reads the data into a buffer. We modify this function to look for sectors in the buffer cache to read before reading from disk via `block_read`. If a cached_block is found with `cached_block->secto`r being the sector we need, we read directly from `cached_block->data`. Otherwise, we call `clock_find_victim` to find a block to evict from the cache, then call `evict_and_replace` to switch out the old cached block with a new block containing the sector we need.
Synchronization:
Before we call `clock_find_victim`, we must acquire the cache_lock. We must release it after calling `evict_and_replace`. Also, we acquire and release the cache block's block_lock before and after reading from `cached_block->data`.

`inode_write_at`
This function iterates through each sector that contains the inode's data and writes data from a buffer into the inode. We modify this function to look for sectors in the

buffer cache to write into before writing into disk via `block_write`. Similarly to `inode_read_at`, we utilize the clock algorithm to evict and replace old blocks if we can't find the block we need in cached_blocks. After the new data has been loaded, we then perform the write to `cached_block→data`. However, when writing we also set the `dirty` bit of the cached_block to 1, indicating that the cached block has been written to.

Synchronization:

Before we call `clock_find_victim`, we must acquire the cache_lock. We must release it after calling `evict_and_replace`. Also, we acquire and release the cache block's block_lock before and after writing from `cached_block→data`.

`filesys_done`

This function will iterate through the cached_blocks array and flush the dirty blocks as follows. For every cached block with dirty=1, we will call `block_write` to write from block→data to the sector corresponding to `block→sector` within the `fs_device` block. Finally, we will null out every entry of the array.

`clock_find_victim`

This new function uses the clock algorithm to pick a block to be evicted from the buffer cache.

Firstly, we check if the cached_blocks array has any null elements i.e. if the cache is not full yet.

If not, we will utilize `buffer_cache→clock_hand` to index into the array of cached blocks, continually turning the clock by incrementing `clock_hand` and searching the list until we find a block with `use=0`. As we encounter blocks with `use=1`, we will decrement use to `0`. We will return the clock_hand's position.

`evict_and_replace`

This function takes in the clock_hand's position (pointing to a cached block) and the sector to load in from disk. We check if the old cached block is dirty; if so, we call `block_write` to write back to the disk. We then update the `cached_block`'s sector, use, and dirty attributes. Finally, we call `block_read` to read from the new sector into cached_block→data.

`timer_interrupt`

We modify this function to flush our cache periodically every N timer interrupts (where N will be chosen after some experimentation). Similarly to `filesys_done`, for every cached block with dirty=1, we will call `block_write` to write from block→data to the sector corresponding to block→sector within the `fs_device` block.

Synchronization

We removed the single global lock to allow threads to read and write to different blocks simultaneously while avoiding race conditions and keeping the buffer cache's state consistent.

We will use a lock around the `buffer_cache` for evictions and replacements, but not during reads and writes to blocks, so that threads can read concurrently to files and write concurrenctly to independent files (synchronization for writing is implemented elsewhere). This global lock is needed so that there are no synchronization issues when running the clock algorithm (i.e. if two threads are simultaneously updating the use bits and looking for free blocks). It is also needed so that threads will not try to access blocks that are being evicted or loaded in.

We use locks around each cache block to allow threads to read and write to different blocks while avoiding race conditions when accessing the same block.

Rationale

Overall, our implementation is time and space efficient, prevents race conditions, and manages the cache well. Iterating through the buffer cache to decide if a block is already cached is `O(64)`, or constant time, so buffer cache accesses are pretty fast. We have two different types of locks (`cache_lock` and `block_lock`) to prevent race conditions while allowing for simultaneous reading /writing to different blocks in the cache. We no longer store `inode_disk→data` in memory to limit the number of cached blocks to 64. Finally, we flush the cache periodically so that the system is less unstable w.r.t. sudden crashes.

Extensible Files

Data Structures and Functions

Inode.c

```
struct inode_disk {
  block_sector_t direct[12];  // The pointers to data blocks. 48 bytes total
  block_sector_t indirect;  //Indirect block pointer. 4 bytes
  block_sector_t doubly_indirect;  // Doubly indirect block pointer. 4 bytes
  off_t length;  // File Size. 4 bytes
  unsigned magic;  // Magic number. 4 bytes
  bool is_dir; //For #Subdirectories usage. 4 bytes
  uint32_t unused[111]; // Padding to make it 512 bytes. 444 bytes
}

struct inode {
  struct list_elem elem;  /* Element in inode list. */
  block_sector_t sector;  /* Sector number of disk location. */
  int open_cnt;           /* Number of openers. */
  bool removed;           /* True if deleted, false otherwise. */
  int deny_write_cnt;     /* 0: writes ok, >0: deny writes. */
  struct lock inode_lock; //Lock for syncing access within the inode itself
}

static struct list open_inodes;
struct lock open_inodes_lock;  // Lock for synchronize access for the open_inodes list
struct lock free_map_lock; //Lock for free map operations
```

We also remove struct inode_disk data from the inode structure as that is redundant due to our new design and using that violates the 64-block limit(via FAQ). We can simply use the `inode->sector` to access the `inode_disk` data or through buffer cache.

Algorithms

`inode_resize`

We create the `inode_resize` function which helps extend and shrink files by either allocating or freeing blocks. When we extend a file, we need to calculate the number of additional blocks required and allocate them from the free map, updating all the appropriate pointers.

The `inode_resize` function extends or shrinks files by allocating or freeing blocks. When extending a file, the algorithm calculates the number of additional blocks required, allocates them from the free map, and updates the appropriate pointers. When changing direct blocks, the pointers can be updated directly while if indirect or doubly direct pointers need to be updated, the function will go through the block pointed to by the indirect pointer(s) to locate the data blocks and update them. Moreover all these newly allocated blocks will be zero-filled unless their allocation is deferred. This zero-filling will ensure that applications see a consistent view of the file, with all uninitialized data set to zero and preventing the risk of exposing residual data.

Moreover, if either extension or reduction would fail at any point, a rollback mechanism ensures consistency by deallocating all newly allocated blocks and resetting the inode structure to its original state.

Changes to existing functions

Accessing files for `inode_write_at` and `inode_read_at`

The `inode_read_at` and `inode_write_at` functions are also modified for when they need to access files. When accessing, they first need to check the direct pointers and then go through the indirect and double indirect pointers list if working with bigger files so they can access and update the appropriate blocks efficiently.

`inode_create`

This function has to be modified to initialize the new inode's `inode_disk` structure correctly, properly intializing the file size, initializing all direct, indirect, and doubly indirect pointers to null/0, and setting the `magic` field. The function ensures that the new inode is ready for file operations and performs cleanup if any errors occur during initialization, maintaining the consistency of the file system.

`inode_write_at`
The `inode_write_at` function needs to be updated to handle file writes that extend beyond the current EOF. If there exists a write which extends the current EOF, we need to call on `inode_resize` to help allocate new blocks as needed.

Synchronization

For synchronization we decide to implement 3 different locks, the `inode_lock`, the `open_inodes_lock` and the `free_map_lock` to supply the requirement of specialized locks in the project specifications.

The `inode_lock` is located in each `inode` and helps synchronize operations such as reading, writing, and resizing. This lock ensures that only one thread can modify the inode's metadata such as `length` and even `deny_write_cnt` at a time. It also helps serialize resize operations, preventing readers from accessing uninited blocks when multiple resizes are taking place.

The `open_inodes_lock` is the global lock that synchronizes access to the `open_inodes` list. This list tracks all currently open `inodes` in the system and can be called to synchronize when adding, removing or indexing into `inodes`

The `free_map_lock` helps control the syncing access to the free map, tracking the allocation status of disk blocks. We utilize it to prevent race conditions during block allocation and deallocation, so that multiple tasks do not modify the same block entry simultaneously. Hence the the `inode_resize` and `inode_create` functions would attempt to acquire on the `free_map_lock` when allocating new blocks.

Rationale

Overall, our structure provides a robust solution for extensible files. The use of direct, indirect, and doubly indirect pointers are a proper mechanism for managing file blocks with direct helping access to smaller files while the other 2 can easily support the 8MiB files we need to support all while raining the 512 byte block size limits. We also are able to help supporting in extending and shrinking file sizes through our implementation. Moreover, we have robust synchronization support with specialized locks helping prevent race conditions.

Subdirectories

Data Structures and Functions

```
struct process {
  // Excluding all attributes from prev projects
  dir* work_dir;
  bool is_root;
}

struct inode_disk {
  // Excluding all attributes from prev projects
  bool is_dir;
}

struct file_descriptor {
  // Excluding all attributes from prev projects
  bool is_dir;
  dir* directory
```

As for functions,

```
bool dir_lookup(const struct dir* dir, const char* name, struc
t inode** inode)

bool dir_add(struct dir* dir, const char* name, block_sector_t
inode_sector)

bool dir_remove(struct dir* dir, const char* name)

bool dir_readdir(struct dir* dir, char name[NAME_MAX + 1])

static int get_next_part(char part[NAME_MAX + 1], const char**
srcp)
```

Algorithms

I’ll start with functions that need modifications:

`sys_open`
We need to handle normal files and directories now. If its a file, use the normal procedure. If its a directory, first use `get_next_part` and `dir_lookup` recursively to traverse into the correct directory and then update the working directory in the PCB.

`sys_read`
We just have to deny reads on directories here. Simply return `-1`.

`sys_write`
Same as `sys_read`, just return `-1` if the input corresponds to a directory.

`sys_exec`
adjust `process_execute` so that child processes inherit their parent process’s current working directory.

`sys_close`
if the input is a file, then proceed as normal. If its a directory, then call `dir_remove`.

`sys_remove`
We’re going to have to adjust the remove syscall to handle deletion of directories. Only empty directories can be deleted (excluding the root directory). if a directory only has `.` or `..` it is still considered empty. We can use the open_cnt attribute in the inode struct to determine when we need to allow or disallow. We will also allow for directories to be removed while they are opened. To prevent any strange behavior though, we will ensure that the the removed attribute of the inode struct is updated on removal, and checked for every file access. If removed is true, file opens and creates will be dissallowed.

next we move onto the main syscalls of the task:

`chdir`
check to see if input should be interpreted as absolute directory or relative directory by recusively calling `dir_lookup` and `get_next_part` and seeing if a file exists in

either interpretation. If not, return false. If it does, then open the inode and set the PCB's `work_dir` to the inode's dir struct.

`mkdir`
Create a new directory with the `dir_add` funciton and create a new inode struct as well. Somewhere during the process add 2 files to the directory `.` and `..` which will point to the working directory and the parent directory respectively.

`readdir`
Check if the file passed in is a directory with `file_descriptor.is_dir`. If yes, then call `dir_readdir` and return `true` on success and `false` otherwise

`isdir`
Check the file_descriptor table for the files `file_descriptor.is_dir` attribute. If its true, return `true` otherwise, return `false`

`inumber`
Use the correct inode retreival function based on whether the fd points to a directory or a normal file. Once the inode is retrieved, return its sector number.

One thing to note here, is that every single syscall will start by determining whether the file path provided as input is an absolute path or a relative path by recursively calling `get_next_part` and `dir_lookup` and checking to see which interpretation of the input path yields our intended file.

Synchronization

For syncronization, anytime we are to edit a directory, we must acquire the `inode_lock` lock initialized in the Extensible Files section to prevent race conditions. This way, we prevent threads from creating files with the same names, removing the same file twice, etc.

Rationale

The implementation of subdirectories enhances the file system's ability to support hierarchical organization, allowing users to logically group files for better organization and navigation. By using recursive calls to `dir_lookup` and `get_next_part`, the system efficiently resolves both relative and absolute paths. Differentiation between files and directories is achieved through the `is_dir` attribute, enabling directory-specific behaviors such as denying `read` and `write` operations while supporting content listing via `readdir`. Directories can only be deleted when empty, and the `removed` attribute ensures consistent handling of deleted directories to prevent race conditions. Modifications to `sys_exec`ensure child processes inherit the parent's working directory, maintaining consistency across processes. The addition of `.`and `..` entries supports scalable nested directory structures, enabling intuitive navigation. Thread safety is maintained through locks, ensuring robust synchronization for operations like creation, deletion, and modification of directories and files. This design provides a scalable, efficient, and user-friendly implementation of subdirectories.

Concept check

Write-behind: To achieve this, we can create a dedicated background thread that periodically scans the cache for a list of dirty blocks and flushes. This thread checks for dirty blocks at fixed intervals. Or, when the cache is close to full capacity, it writes back some of the dirty blocks first. Dirty blocks are written to disk in a batch to improve write efficiency. The marking of dirty blocks is cleared after a successful write.

Read-ahead: To achieve this, we set a fixed number of pre-read blocks N. Assuming that the block currently accessed by the user is current_block, then our asynchronous I/O will pre-load current_block+1 to current_block+N blocks into the cache. If the cache space is insufficient, some infrequently used or low-priority blocks can be evicted (according to the cache replacement policy, e.g., LRU). This way, once future blocks are loaded into the cache buffer, when the user actually needs to read those blocks, he can get the data directly from the cache without waiting for a disc read.